МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий

Кафедра Программной инженерии

Специальность 1-40 01 01 Информационные технологии

Специализация Конструирование программного обеспечения

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:**

«Разработка компилятора KVV-2024»

Выполнил студент Казакова Виктория Васильевна

(Ф.И.О.)

Руководитель проекта асс. Гончар Е.А

(учен. степень, звание, должность, подпись, Ф.И.О.)

Заведующий кафедрой к.т.н., доц. Смелов В.В.

(учен. степень, звание, должность, подпись, Ф.И.О.)

Консультанты асс. Гончар Е.А.

(учен. степень, звание, должность, подпись, Ф.И.О.)

Нормоконтролер асс. Гончар Е.А.

(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой

Минск 2024

**Содержание**

[Введение 5](#_Toc185085676)

[Глава 1. Спецификация языка программирования 6](#_Toc185085677)

[1.1 Характеристика языка программирования 6](#_Toc185085678)

[1.2 Определение алфавита языка программирования 6](#_Toc185085679)

[1.3 Применяемые сепараторы 7](#_Toc185085680)

[1.4 Применяемые кодировки 7](#_Toc185085681)

[1.5 Типы данных 8](#_Toc185085682)

[1.6 Преобразование типов данных 8](#_Toc185085683)

[1.7 Идентификаторы 8](#_Toc185085684)

[1.8 Литералы 8](#_Toc185085685)

[1.9 Объявление данных 9](#_Toc185085686)

[1.10 Инициализация данных 9](#_Toc185085687)

[1.11 Инструкции языка 10](#_Toc185085688)

[1.12 Операции языка 10](#_Toc185085689)

[1.13 Выражения и их вычисления 11](#_Toc185085690)

[1.14 Конструкции языка 12](#_Toc185085691)

[1.15 Область видимости идентификаторов 12](#_Toc185085692)

[1.16 Семантические проверки 12](#_Toc185085693)

[1.17 Распределение оперативной памяти на этапе выполнения 13](#_Toc185085694)

[1.18 Стандартная библиотека и её состав 13](#_Toc185085695)

[1.19 Ввод и вывод данных 14](#_Toc185085696)

[1.20 Точка входа 14](#_Toc185085697)

[1.21 Препроцессор 14](#_Toc185085698)

[1.22 Соглашения о вызовах 14](#_Toc185085699)

[1.23 Объектный код 14](#_Toc185085700)

[1.24 Классификация сообщений транслятора 14](#_Toc185085701)

[1.25 Контрольный пример 15](#_Toc185085702)

[Глава 2 Структура транслятора 16](#_Toc185085703)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 16](#_Toc185085704)

[2.2 Перечень входных параметров транслятора 17](#_Toc185085705)

[2.3 Протоколы, формируемые транслятором 17](#_Toc185085705)

[Глава 3 Разработка лексического анализатора 18](#_Toc185085706)

[3.1 Структура лексического анализатора 18](#_Toc185085707)

[3.2 Контроль входных символов 19](#_Toc185085708)

[3.3 Удаление избыточных символов 19](#_Toc185085709)

[3.4 Перечень ключевых слов 19](#_Toc185085710)

[3.5 Основные структуры данных 22](#_Toc185085711)

[3.6 Структура и перечень сообщений лексического анализатора 22](#_Toc185085712)

[3.7 Принцип обработки ошибок 22](#_Toc185085713)

[3.8 Параметры лексического анализатора 22](#_Toc185085714)

[3.9 Алгоритм лексического анализа 23](#_Toc185085715)

[Глава 4 Разработка синтаксического анализатора 24](#_Toc185085716)

[4.1 Структура синтаксического анализатора 24](#_Toc185085717)

[4.2 Контекстно-свободная грамматика, описывающая синтаксис языка 24](#_Toc185085718)

[4.3 Построение конечного магазинного автомата 25](#_Toc185085719)

[4.4 Основные структуры данных 26](#_Toc185085720)

[4.5 Описание алгоритма синтаксического разбора 26](#_Toc185085721)

[4.6 Структура и перечень сообщений синтаксического анализатора 27](#_Toc185085722)

[4.7 Параметры синтаксического анализатора 27](#_Toc185085723)

[4.8 Принцип обработки ошибок 27](#_Toc185085724)

[4.9 Контрольный пример 28](#_Toc185085725)

[Глава 5 Разработка семантического анализатора 29](#_Toc185085726)

[5.1 Структура семантического анализатора 29](#_Toc185085727)

[5.2 Функции семантического анализатора 29](#_Toc185085728)

[5.3 Структура и перечень сообщений семантического анализатора 29](#_Toc185085729)

[5.4 Принцип обработки ошибок 30](#_Toc185085730)

[5.5 Контрольный пример 30](#_Toc185085731)

[Глава 6 Вычисление выражений 31](#_Toc185085732)

[6.1 Выражения, допускаемые языком 31](#_Toc185085733)

[6.2 Польская запись и принцип ее построения 31](#_Toc185085734)

[6.3 Программная реализация обработки выражений 32](#_Toc185085735)

[6.4 Контрольный пример 32](#_Toc185085736)

[Глава 7 Генерация кода 33](#_Toc185085737)

[7.1 Структура генератора кода 33](#_Toc185085738)

[7.2 Представление типов данных в оперативной памяти 33](#_Toc185085739)

[7.3 Статическая библиотека 34](#_Toc185085740)

[7.4 Особенности алгоритма генерации кода 34](#_Toc185085741)

[7.5 Входные параметры, управляющие генерацией кода 35](#_Toc185085742)

[7.6 Контрольный пример 35](#_Toc185085743)

[Глава 8 Тестирование транслятора 36](#_Toc185085744)

[8.1 Общие положения 36](#_Toc185085745)

[8.2 Результаты тестирования 36](#_Toc185085746)

[Глава 9 Разработка и тестирование интерпретатора 38](#_Toc185085747)

[9.1 Структура и перечень сообщений времени выполнения 38](#_Toc185085748)

[Заключение 39](#_Toc185085749)

[Список использованных источников 40](#_Toc185085750)

[Приложение А 41](#_Toc185085751)

[Приложение Б 43](#_Toc185085752)

[Приложение Г 47](#_Toc185085753)

[Приложение Д 50](#_Toc185085754)

[Приложение Е 51](#_Toc185085755)

[Приложение Ж 53](#_Toc185085756)

[Приложение З 54](#_Toc185085757)

Введение

Целью курсового проекта поставлена задача разработки компилятора для моего языка программирования KVV-2024. Этот язык программирования предназначен для выполнения простейших операций и арифметических действий над числами.

Компилятор KVV-2024 – это программа, задачей которого является перевод программы, написанной на языке программирования KVV-2024 в программу на язык ассемблера.

Транслятор KVV-2024 состоит из следующих частей:

– лексический и семантический анализаторы;

– синтаксический анализатор;

– генератор исходного кода на языке ассемблера.

Исходя из цели курсового проекта, были определены следующие задачи:

– разработка спецификации языка программирования;

– разработка структуры транслятора;

– разработка лексического и семантического анализаторов;

– разработка синтаксического анализатора;

– преобразование выражений;

– генерация кода на язык ассемблер;

– тестирование транслятора.

Решения каждой из поставленных задач будут приведены в соответствующих главах курсового проекта.

Глава 1. Спецификация языка программирования

## **Характеристика языка программирования**

Язык программирования KVV-2024 является процедурным, строго типизированным, не объектно-ориентированным, компилируемым.

Процедурный язык программирования — язык высокого уровня, в котором используется метод разбиения программ на отдельные связанные между собой модули — подпрограммы (процедуры и функции).

Строго типизированный язык программирования — язык, в котором переменные привязаны к конкретным типам данных. Язык не позволяет смешивать в выражениях различные типы и не выполняет автоматические неявные преобразования.

Объектно-ориентированный язык программирования — язык, построенный на принципах объектно-ориентированного программирования. В основе концепции объектно-ориентированного программирования лежит понятие объекта — некой сущности, которая объединяет в себе поля (данные) и методы (выполняемые объектом действия).

Компилируемый язык программирования — язык программирования, исходный код которого преобразуется компилятором в исходный код на другом языке программирования.

* 1. **Определение алфавита языка программирования**

Алфавит языка KVV-2024 основан на кодировке Windows-1251, представленной на рисунке 1.1.



Рисунок 1.1 – Алфавит входных символов

На этапе выполнения могут использоваться символы латинского алфавита, кириллица нижнего регистра, цифры десятичной системы счисления от 0 до 9, а также цифры шестнадцатеричной системы счисления, спецсимволы, а также непечатные символы пробела, табуляции и перевода строки.

## **Применяемые сепараторы**

Символы, которые являются сепараторами представлены в таблице 1.1.

Таблица 1.1 – Сепараторы

|  |  |  |
| --- | --- | --- |
| Сепаратор | Название | Область применения |
| ‘ ‘ | Пробел | Допускается везде, кроме идентификаторов и ключевых слов |
| ; | Точка с запятой | Разделение конструкций |
| {…} | Фигурные скобки | Заключение программного блока |
| (…) | Круглые скобки | Приоритет операций, параметры функции |
| […] | Квадратные скобки | Заключение блоков if, else, while |
| ‘…’ | Одинарные кавычки | Допускается везде, кроме идентификаторов и ключевых слов |
| = | Знак «равно» | Присваивание значения |
| , | Запятая | Разделение параметров |
| ~  !  >  <  »  « | Знаки «равенство», «неравенство», «больше», «меньше», «больше или равно» , «меньше или равно» | Логические выражения |
| +  -  \*  /  % | Знаки «сумма», «разность», «умножение», «деление», «деление по модулю» | Арифметические выражения |

Сепараторы – это программные конструкции, служащие для разделения блоков кода.

## **Применяемые кодировки**

Для написания исходного кода на языке программирования KVV-2024 используется кодировка Windows-1251.

Windows-1251 – набор символов и кодировка, являющаяся 8-битной кодировкой для русских версий Windows.

## **Типы данных**

В языке KVV-2024 реализованы два типа данных: беззнаковый целочисленный и строковый. Описание реализованных типов данных представлено в таблице 1.2.

|  |  |
| --- | --- |
| Тип данных | Описание типа данных |
| Целочисленный беззнаковй uint | Фундаментальный тип данных. Используется для работы с целочисленными значениями. В памяти занимает 4 байта.  Максимальное значение: 4,294,967,295.  Минимальное значение: 0.  Инициализация по умолчанию: значение 0. |
| Строковый тип данных string | Фундаментальный тип данных. Используется для работы с символами, каждый символ в памяти занимает 1 байт.  Максимальное количество символов: 255.  Инициализация по умолчанию: длина 0, символ конца строки “\0”. |

Таблица 1.2 – Типы данных языка KVV-2024

Тип данных — это характеристика, определяющая свойства данных и операции, которые могут быть выполнены над ними. В программировании тип данных указывает, какого рода значения может принимать переменная, константа или выражение. Типы данных являются фундаментальной концепцией для работы с данными, поскольку они определяют структуру и поведение данных в программе.

## **Преобразование типов данных**

Преобразование типов данных не поддерживается, т.е. язык является строготипизированным.

## **Идентификаторы**

В имени идентификатора допускаются символы латинского алфавита и кириллица в нижнем регистре. Максимальная длина имени идентификатора - 30 символов. Максимальная длина имени идентификатора функции - 30 символов. При вводе идентификатора длиной более разрешенного количества символов, он будет усекаться. Имя идентификатора не может совпадать с ключевыми словами и не может иметь имя, как функция, уже содержащаяся в стандартной библиотеке.

## **Литералы**

С помощью литералов осуществляется инициализация переменных. В языке KVV-2024 существует два типа литералов: целые и строковые.

Литералы – представляют собой константные значения определённого типа, такие как числа, строки, символы и логические значения.

Краткое описание литералов языка KVV-2024 представлено в таблице 1.3.

|  |  |  |  |
| --- | --- | --- | --- |
| Тип литерала | Регулярное выражение | Описание | Пример |
| Целочисленный литерал | [1-9]+[0-9]\* | Целочисленные литералы,по умолчанию инициализируются 0. Литералы могут быть только rvalue. | declare uint sum;  sum = 15;  15 – целочисленный литерал. |
| Строковые литерал | [a-z|а-я|0-9|']+ | Символы, заключённые в '…' (одинарные кавычки), по умолчанию инициализируются пустой строкой. Литералы могут быть только rvalue. | declare string text = ‘text’;  text – строковый литерал. |

Таблица 1.3 – Описание литералов

rvalue - это то, что можно только присваивать, например, литералы или ре-зультаты выражений.

## **Объявление данных**

Область видимости «сверху вниз» (по принципу С++). В языке KVV-2024 требуется обязательное объявление переменной перед её инициализацией и последующим использованием. Все переменные должны находиться внутри программного блока. Имеется возможность объявления одинаковых переменных в разных блоках, т. к. переменные, объявленные в одной функции, недоступны в другой. Каждая переменная получает префикс – название функции, в которой она объявлена.

Объявление переменных осуществляется с помощью ключевого слова declare. Для объявления функций используется ключевое слово function, перед которым указывается тип функции, после которого имя функции. Далее обязателен список параметров и тело функции.

## **Инициализация данных**

При объявлении переменной не допускается инициализация данных. Краткое описание способов инициализации переменных языка KVV-2024 представлено в таблице 1.4.

Таблица 1.4 – Способы инициализации переменных

|  |  |  |
| --- | --- | --- |
| Конструкция | Описание | Пример |
| declare <тип данных> <идентификатор>; | Автоматическая инициализация: переменные типа uint инициализируются нулём, переменные типа string – пустой строкой. | declare uint counter;  declare string message; |

Окончание таблицы 1.4 – Способы инициализации переменных

|  |  |  |
| --- | --- | --- |
| <идентификатор> = <значение>; | Присваивание переменной значения. | Counter = 0;  message = ‘new calculating’; |

Инициализация данных в программировании означает присвоение начальных значений переменным или структурам данных в момент их создания. Это позволяет установить изначальное состояние для использования данных в программе.

## **Инструкции языка**

Все возможные инструкции языка программирования KVV-2024 представлены в общем виде в таблице 1.5.

Таблица 1.5 – Инструкции языка программирования KVV-2024

|  |  |
| --- | --- |
| Инструкция | Запись на языке KVV -2024 |
| Объявление переменной | declare <тип данных> <идентификатор>; |
| Присваивание | <идентификатор> = <значение>/<идентификатор>; |
| Объявление внешней функции | <тип данных> function <идентификатор> (<тип данных> <идентификатор>, …) |
| Блок инструкций | main  {  …  }; |
| Возврат из подпрограммы | return <идентификатор> / <литерал>; |
| Условная инструкция | if (условие) [/программный блок если условие верно/]  else [/программный блок если условие ложно/]; |
| Вывод данных | write (<идентификатор> / <литерал>); |

Таким образом язык KVV-2024 использует следующий синтаксис, приведённый выше.

* 1. **Операции языка**

Язык программирования KVV-2024 может выполнять операции сравнения, представленные в таблице 1.6.

Таблица 1.6 – Операции языка программирования KVV-2024

|  |  |  |  |
| --- | --- | --- | --- |
| Операция | Примечание | Типы данных | Пример |
| < | Знак «меньше» для условной инструкции | (uint, uint) | if(operation < 5) […] |

Окончание таблицы 1.6 – Операции языка программирования KVV-2024

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| > | | Знак «больше» для условной инструкции | (uint, uint) | | if(operation > 6) […] |
| ( | | Приоритет операций | - | | operation = (operation-1) / 3; |
| ) | |
| ~ | Равенство | | | (uint, uint) | a ~ b; |
| ! | Неравенство | | | (uint, uint) | a ! b; |
| « | Меньше или равно | | | (uint, uint) | a « b; |
| » | Больше или равно | | | (uint, uint) | a » b; |
| + | Сумма | | | (uint, uint) | a + b; |
| - | Разность | | | (uint, uint) | a – b; |
| \* | Умножение | | | (uint, uint) | a \* b; |
| / | Деление | | | (uint, uint) | a / b; |
| % | Деление по модулю | | | (uint, uint) | a % b; |

Приоритет подчиняется правилам:

* если в выражение без скобок входят только сложение и вычитание или только умножение, деление и остаток от деления, то действия выполняют в том порядке, в каком они написаны;
* если в выражение без скобок входят не только действия сложения и вычитания, но и умножения, деления и остатка от деления, или оба этих действия, то сначала выполняется умножениея, деление и остаток от деления, а затем сложение и вычитание;
* если в выражении имеются скобки, то сначала вычисляют значение выражений в скобках.

## **Выражения и их вычисления**

Вычисление выражений – комбинация значений или переменных, констант, переменных, операций и функций, которая может быть интерпретирована в соответствии с правилами конкретного языка. Всякое выражение составляется согласно следующим правилам:

* допускается использовать скобки для смены приоритета операций;
* выражение записывается в строку без переносов;
* допускается использовать в выражении вызов функции, вычисляющей и возвращающей целочисленное значение.
* просмотр выражения идет слева направо;
* выражение считывается до символа сепаратора точки с запятой.
* использование в одном выражении операндов разных типов недопустимо;

Перед генерацией кода каждое выражение приводится к записи в польской нотации для удобства дальнейшего вычисления выражения на языке ассемблера.

## **Конструкции языка**

Ключевые программные конструкции языка программирования KVV-2024 представлены в таблице 1.7.

Таблица 1.7 – Программные конструкции языка KVV-2024

|  |  |
| --- | --- |
| Конструкция | Запись на языке KVV-2024 |
| Главная функция (точка входа) | main  {  …  return <идентификатор> / <литерал>;  }; |
| Функция | function <тип> <идентификатор> (<тип> <идентификатор>, …)  {  …  return <идентификатор> / <литерал>;  }; |

Конструкция языка — это элемент программы, который соответствует синтаксическим правилам языка программирования. Она состоит из ключевых слов, операторов, выражений, литералов или других компонентов языка, используемых для выполнения определённых задач, таких как объявление переменных, создание условий, циклов или вызов функций.

## **Область видимости идентификаторов**

В языке KVV-2024 все переменные являются локальными. Они обязаны находится внутри программного блока функций (по принципу С++). Объявление глобальных переменных не предусмотрено. Объявление пользовательских областей видимости не предусмотрено.

## **Семантические проверки**

Таблица с перечнем семантических проверок, предусмотренных языком, приведена в таблице 1.8.

Таблица 1.8 – Семантические проверки

|  |  |
| --- | --- |
| Номер | Правило |
| 1 | В функции отсутствует return |
| 2 | Функция не всегда возвращает значение |
| 3 | Попытка объявить функцию в main |
| 4 | Ошибка в блоке main: функция возвращает не целочисленное занчение |
| 5 | Тип возвращаемого значения не соответствует типу функции |
| 6 | Слишком много аргументов передано в функцию |
| 7 | Тип передаваемого аргумента не соответствует типу параметра функции |

Окончание таблицы 1.8 – Семантические проверки

|  |  |
| --- | --- |
| 8 | Слишком мало аргументов передано в функцию |
| 9 | Попытка переопределить библиотечную функцию |
| 10 | Неопознаая библиотечная функция |

Следовательно, при несоблюдении хотя бы одного из правил, семантический анализатор будет сообщать об ошибке.

## **Распределение оперативной памяти на этапе выполнения**

При трансляции кода используется две области памяти: сегмент констант и сегмент данных. Они заполняются с помощью таблицы лексем и таблицы идентификаторов, сформированных на этапе лексического анализа. В сегмент констант заносятся строковые и целочисленные литералы. В сегмент данных – переменные и параметры функций.

## **Стандартная библиотека и её состав**

Стандартная библиотека KVV-2024 написана на языке программирования C++. Функции стандартной библиотеки с описанием представлены в таблице 1.9.

Таблица 1.9 – Состав стандартной библиотеки

|  |  |  |
| --- | --- | --- |
| Функция(C++) | Возвращаемое значение | Описание |
| unsigned int toPow(unsigned int a, unsigned int b) | uint | Функция возведения числа в степень |
| unsigned int random(unsigned int a) | uint | Функция вывода на консоль строкового идентификатора/литерала |
| int writenum(unsigned int value) | uint | Функция выводит числовое значение value в стандартный поток вывода (консоль). |
| Int writestr(char\* ptr) | uint | Функция выводит строку ptr в стандартный поток вывода (консоль). Если ptr равен nullptr, ничего не выводит. |
| Unsigned int writenumline(unsigned int value) | uint | Функция выводит числовое значение value в стандартный поток вывода (консоль) с переходом на новую строку. |
| Unsigned int writestrline(char\* ptr) | uint | Функция выводит строку ptr в стандартный поток вывода (консоль) с переходом на новую строку. Если строка пустая, выводит просто пустую строку с переходом на новую строку. |

Стандартная библиотека написана на языке С++, подключается к транслированному коду на этапе генерации кода. Вызовы стандартных функций доступны там же, где и вызов пользовательских функций. Также в стандартной библиотеке реализованы функции для манипулирования выводом, недоступные конечному пользователю. Для вывода предусмотрен оператор writeиwriteline.

## **Ввод и вывод данных**

В языке KVV-2024 не реализованы средства ввода данных.

Для вывода данных в стандартный поток вывода предусмотрен оператор write, который входят в состав стандартной библиотеки и описан в таблице 1.9.

## **Точка входа**

В языке KVV-2024 каждая программа должна содержать главную функцию main, т. Е. точку входа, с которой начнется последовательное выполнение программы.

## **Препроцессор**

Препроцессор в языке программирования KVV-2024 не предусмотрен.

## **Соглашения о вызовах**

В языке вызов функций происходит по соглашению о вызовах stdcall. Особенности stdcall:

– все параметры функции передаются через стек;

– память высвобождает вызываемый код;

– занесение в стек параметров идёт справа налево.

## **Объектный код**

KVV-2024 транслируется в язык ассемблера.

## **Классификация сообщений транслятора**

В случае возникновения ошибки в коде программы на языке KVV-2024 и выявления её транслятором в текущий файл протокола выводится сообщение. Их классификация сообщений приведена в таблице 1.10.

Таблица 1.10 – Классификация сообщений транслятора

|  |  |
| --- | --- |
| Интервал | Описание ошибок |
| 0-99 | Системные ошибки |
| 100-109 | Ошибки параметров |
| 110-119 | Ошибки открытия и чтения файлов |

Окончание таблицы 1.10 – Классификация сообщений транслятора

|  |  |
| --- | --- |
| 120-129 | Ошибки лексического анализа |
| 600-699 | Ошибки синтаксического анализа |
| 700-800 | Ошибки семантического анализа |

Обрабатываются ошибки на всех этапах обработки исходного кода, то есть во время прохождения различных этапов анализа.

* 1. Контрольный пример

Контрольный пример демонстрирует главные особенности языка KVV-2024: его фундаментальные типы, основные структуры, функции, процедуры, использование функций статической библиотеки. Исходный код контрольного примера представлен в приложении А.

Глава 2 Структура транслятора

2.1 Компоненты транслятора, их назначение и принципы взаимодействия

Трансляторы играют ключевую роль в разработке программного обеспечения, обеспечивая перевод программ из одного языка программирования в другой. В случае языка KVV-2024, транслятор выполняет преобразование программы, написанной на KVV-2024, в программу на языке ассемблера. Компоненты, составляющие транслятор, включают лексический, синтаксический и семантический анализаторы, а также генератор кода на языке ассемблера. Их взаимодействие основывается на определенных принципах и правилах, обеспечивающих корректность и точность процесса трансляции. Принцип их взаимодействия представлен на рисунке 2.1.

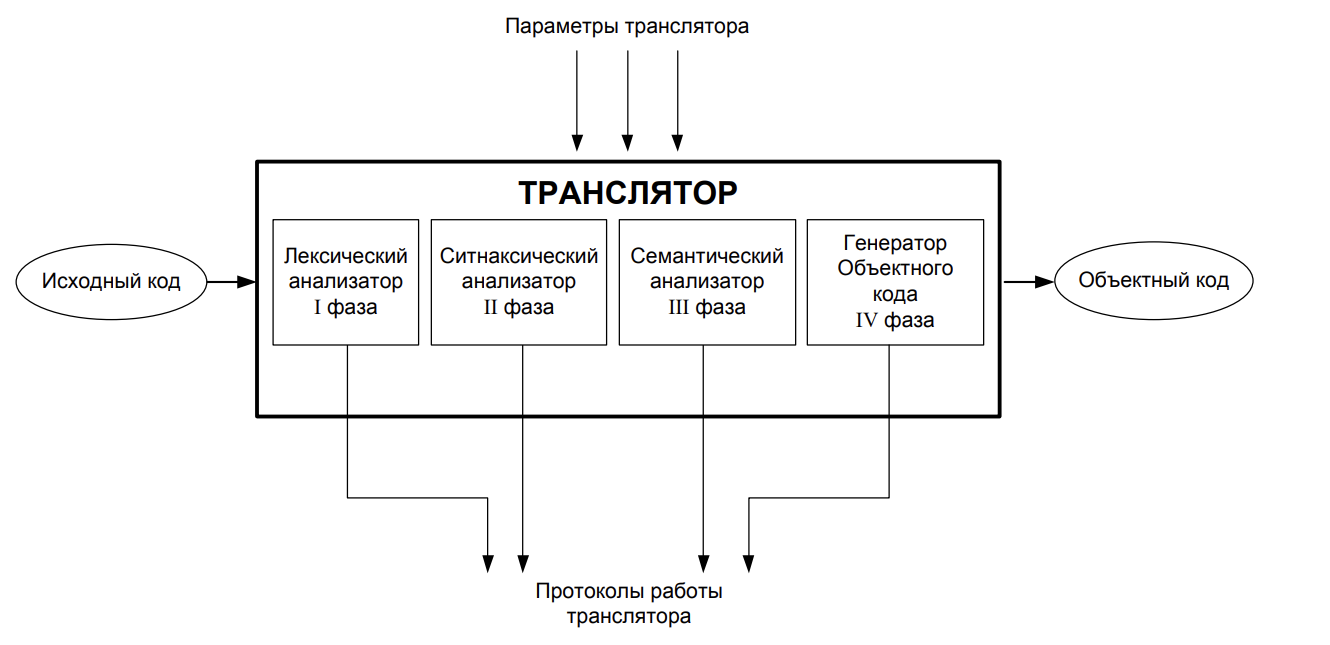


Рисунок 2.1 – Структура транслятора

Транслятор – это программа преобразующая исходный код на одном языке программирования в исходный код на другом языке программирования.

Лексический анализатор – первая фаза трансляции, принимает на вход уже первично обработанный и разбитый на отдельные компоненты исходный код на языке KVV-2024. Формирует таблицу идентификаторов и таблицу лексем, а также занимается обнаружением ошибок, связанных с лексикой языка.

Синтаксический анализатор – принимает на вход таблицу лексем, сформированную лексическим анализатором. Перебирая каждое правило языка (допустимую конструкцию) он выявляет синтаксические ошибки, допущенные в исходном коде. Формирует дерево разбора, а также выводит трассировку (разбор) цепочек.

Семантический анализатор – состоит из нескольких функций, отвечающих за выявления тех или иных ошибок, а также некоторых проверок, выполняемых на этапе лексического анализатора. В зависимости от задачи функции на ее вход подается таблица лексем либо таблица лексем и идентификаторов.

Генератор кода – принимает на вход таблицу идентификаторов и таблицу лексем. Задача этого компонента заключается в трансляции, уже пройденного все предыдущие этапы кода на языке KVV-2024, в код на языке Ассемблер.

2.2 Перечень входных параметров транслятора

Входные параметры транслятора представлены в таблице 2.1.

Таблица 2.1 – Входные параметры транслятора языка KVV-2024

|  |  |  |
| --- | --- | --- |
| Входной параметр | Значение по умолчанию | Характеристика |
| -in:<путь к in-файлу > | Не предусмотрено | Входной файл с расширением .txt, в котором содержится исходный код языка KVV-2024 |
| -log:<путь к log-файлу > | <имя in-файла>.log | Файл, содержащий информацию о работе транслятора. |
| -out:<путь к out -файлу > | <имя in-файла>.out | Файл для записи результата работы лексического и синтаксического анализаторов. |

Данные входные параметры транслятора используются для форматирования файлов с результатами работы лексического, синтаксического и семантического анализаторов.

## **2.3 Протоколы, формируемые транслятором**

Таблица с перечнем протоколов, формируемых транслятором языка KVV-2024 и их назначением представлена в таблице 2.2.

Таблица 2.2 Протоколы, формируемые транслятором языка KVV-2024

|  |  |  |
| --- | --- | --- |
| Формируемый протокол | | Описание выходного протокола |
| Файл журнала, заданный параметром \*.out | Содержит таблицу лексем и таблицу идентификаторов, протокол работы синтаксического анализатора, полученные на этапе лексического и синтаксического анализа, а также результат работы алгоритма преобразования выражений к польской записи. | |
| Выходной файл с расширением \*.log | Содержит информацию об ошибках. Содержит информацию про входные параметры, общем количестве символов и строк(исходные данные). | |

Протокол работы нужен для отображения хода выполнения трансляции языка KVV-2024. Благодаря им пользователь может обнаружить некорректно введенные данные или ошибки в исходном коде программы.

Глава 3 Разработка лексического анализатора

3.1 Структура лексического анализатора

Первая стадия работы компилятора называется лексическим анализом, а программа, её реализующая, – лексическим анализатором (сканером). На вход лексического анализатора подаётся последовательность символов входного языка. Лексический анализатор выделяет в этой последовательности простейшие конструкции языка, которые называют лексическими единицами. Примеры лексических единиц: идентификаторы, числа, символы операций, служебные слова и т.д. Лексический анализатор преобразует исходный текст, заменяя лексические единицы их внутренним представлением – лексемами, для создания промежуточного представления исходной программы. Каждой лексеме сопоставляется ее тип и запись в таблице идентификаторов, в которой хранится дополнительная информация. Результатом работы лексического анализатора являются таблица лексем и идентификаторов. Структура лексического анализатора представлена на рисунке 3.1.

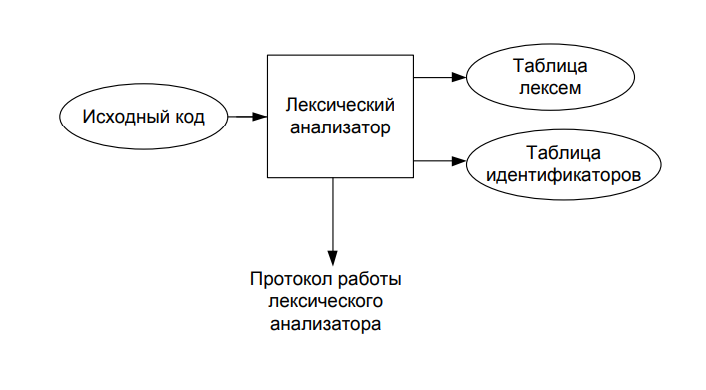


Рисунок 3.1 Структура лексического анализатора KVV-2024

Функции лексического анализатора:

− удаление «пустых» символов и комментариев. Если «пустые» символы (пробелы, знаки табуляции и перехода на новую строку) и комментарии будут удалены лексическим анализатором, синтаксический анализатор никогда не столкнется с ними (альтернативный способ, состоящий в модификации грамматики для включения «пустых» символов и комментариев в синтаксис, достаточно сложен для реализации);

− распознавание идентификаторов и ключевых слов;

− распознавание констант;

− распознавание разделителей и знаков операций.

Лексический анализатор должен выдавать сообщения о наличии во входном тексте ошибок, если они будут обнаружены на этапе лексического анализа.

## **3.2 Контроль входных символов**

Принцип работы таблицы заключается в соответствии значения каждому элементу в шестнадцатеричной системе счисления значению в таблице ASCII.

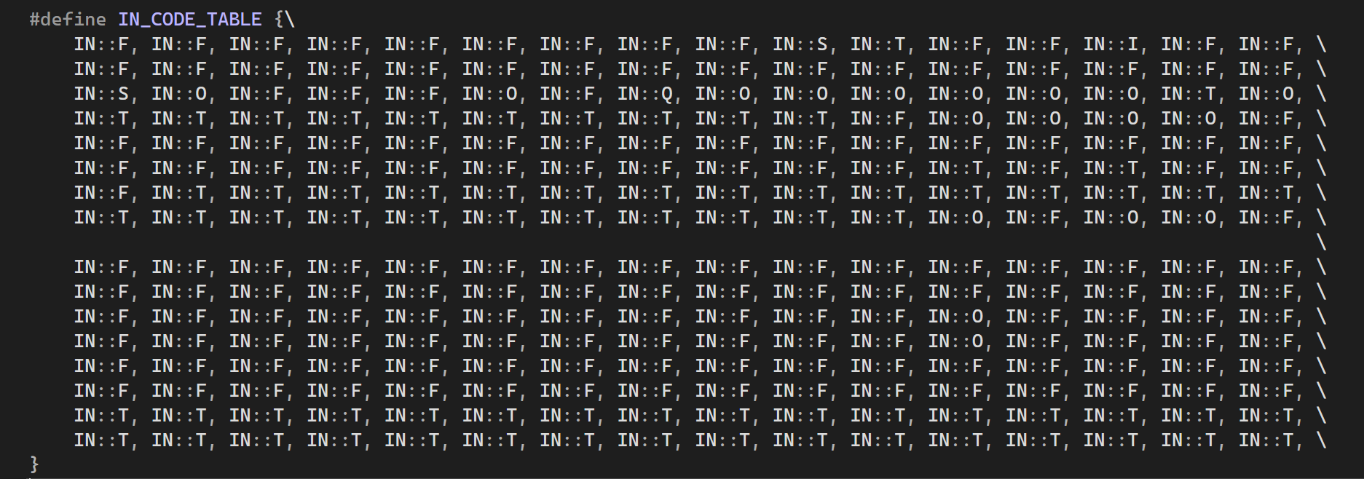


Рисунок 3.2. – Таблица контроля входных символов

Описание значения символов: T – разрешённый символ, F – запрещённый символ, I – игнорируемый символ, O – операция.

## **3.3 Удаление избыточных символов**

Избыточными символами являются символы табуляции и пробелы.

Избыточные символы удаляются на этапе разбиения исходного кода на токены.

Описание алгоритма удаления избыточных символов:

1. Посимвольно считываем файл с исходным кодом программы;

2. Встреча пробела или знака табуляции является своего рода встречей символа-сепаратора;

3. В отличие от других символов-сепараторов не записываем в очередь лексем эти символы, т.е. игнорируем.

Продолжаем посимвольное считывание файла до встречи с символом, отличным от избыточного.

3.4 Перечень ключевых слов

Лексический анализатор преобразует исходный текст, заменяя лексические единицы лексемами для создания промежуточного представления исходной программы.

Лексемы – это символы, соответствующие ключевым словам, символам операций и сепараторам, необходимые для упрощения дальнейшей обработки исходного кода программы. Соответствие токенов и лексем представлено в таблице 3.1.

Ключевые слова в — это зарезервированные слова, которые имеют специальное значение для компилятора. Эти слова нельзя использовать в качестве имен переменных, функций или других пользовательских идентификаторов.

Таблице 3.1 – Таблица соответствий цепочек, их типов и лексем

|  |  |  |
| --- | --- | --- |
| Токен | Тип | Лексема |
| stringln | Ключевое слово | c |
| write | Ключевое слово | p |
| uint | Ключевое слово | n |
| string | Ключевое слово | s |
| Идентификатор | Переменная, функция или параметр | i |
| Целочисленный литерал | Данные | l |
| Строковый  литерал | Данные | l |
| main | Ключевое слово | m |
| function | Ключевое слово | f |
| declare | Ключевое слово | d |
| out | Ключевое слово | p |
| if | Ключевое слово | I |
| else | Ключевое слово | E |
| +  -  \*  /  =  >  <  $  %  ^  & | Операторы | +  -  \*  /  =  >  <  $  %  ^  & |
| ;  ,  {}  [] | Сепараторы | ;  ,  {}  [] |

Пример реализации таблицы лексем представлен в приложении B.

Также в приложении находятся конечные автоматы, соответствующие лексемам языка KVV-2024.

Каждому выражению соответствует детерминированный конечный автомат, по которому происходит разбор данного выражения. На каждый автомат в массиве подаётся токен и с помощью регулярного выражения, соответствующего данному графу переходов, происходит разбор. В случае успешного разбора выражения оно записывается в таблицу лексем. Если выражение является идентификатором или литералом, информация также заносится в таблицу идентификаторов. Структура конечного автомата и пример графа перехода конечного автомата изображены на рисунках 3.3 и 3.4 соответственно.

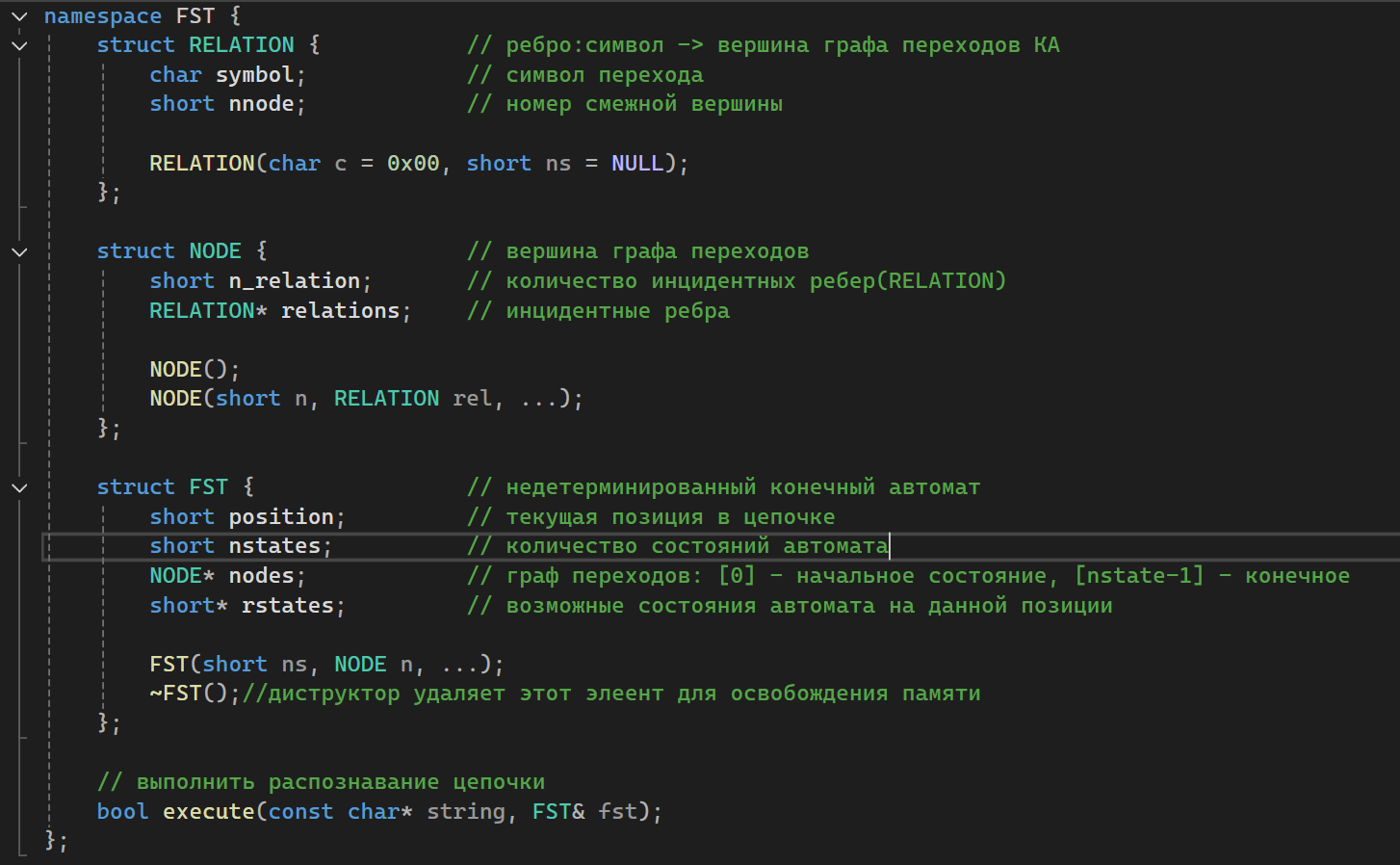


Рисунок 3.3 – Структура конечного автомата

Определение конечного автомата (КА):

КА это пятерка M = (S, I, δ, s0, F) , где:

* S – конечное множество состояний устройства управления;
* I – алфавит входных символов;
* δ – функция переходов, отображающая S × (I ∪{λ}) в множество подмножеств S : δ(s,i) ⊂ S, s∈S, i∈ I ;
* s0 ∈ S – начальное состояние устройства управления;
* F ⊆ S – множество заключительных (допускающих) состояний устройства управления;

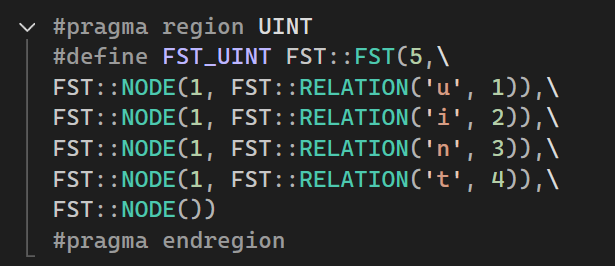


Рисунок 3.4– Пример реализации графа конечного автомата для токена uint

Если δ(s,λ) = ∅ и |δ (s,a)|≤1, то конечный автомат детерминированный (ДКА) иначе – конечный автомат недетерминированный (НКА).

## **3.5 Основные структуры данных**

Основные структуры таблиц лексем и идентификаторов данных языка KVV-2024, используемых для хранения, представлены в приложении Б. В таблице лексем содержится лексема, её номер, полученный при разборе, номер строки в исходном код. В таблице идентификаторов содержится имя идентификатора, номер в таблице лексем, тип данных, смысловой тип идентификатора и его значение.

3.6 Структура и перечень сообщений лексического анализатора

Перечень сообщений лексического анализатора представлен в таблице 3.2.

Таблица 3.2 – ­­­­Перечень сообщений лексического анализатора

|  |  |
| --- | --- |
| Код | Сообщение |
| 120 | Ошибка лексического анализа: превышено максимальное количество строк в таблице лексем |
| 121 | Ошибка лексического анализа: превышена максимальная емкость таблицы лексем |
| 122 | Ошибка лексического анализа: Выход за пределы таблицы лексем |
| 123 | Ошибка лексического анализа: множественное объявление лексемы |
| 124 | Ошибка лексического анализа: идентификатор не объявлен |

Таким образом, при написании кода, достаточно просто понять, с чем конкретно связана та или иная ошибка.

## **3.7 Принцип обработки ошибок**

В случае если в результате работы лексического анализатора найдена ошибка, то анализатор заполняет структуру, содержащую ошибки, произошедшие в процессе лексического анализа.

После чего вызывается функция генерации ошибки, в которую передается, в зависимости от места возникновения ошибки, следующая информация: код ошибки, номер строки в коде, номер позиции в строке или только код ошибки. Пользователь может ознакомиться с данной ошибкой, открыв протокол.

**3.8 Параметры лексического анализатора**

Входные данные для лексического анализатора – это текст программы, который может быть представлен в виде строки или последовательности символов.

Результаты работы лексического анализатора – это последовательность лексем (токенов), которые образуют входную программу.

Результаты работы лексического анализатора, а именно таблицы лексем и идентификаторов выводятся в файл журнала.

**3.9 Алгоритм лексического анализа**

* проверяет входной поток символов программы на исходном языке на допустимость, удаляет лишние пробелы и добавляет сепаратор для вычисления номера строки для каждой лексемы;
* для выделенной части входного потока выполняется функция распознавания лексемы;
* при успешном распознавании информация о выделенной лексеме заносится в таблицу лексем и таблицу идентификаторов, и алгоритм возвращается к первому этапу;
* формирует протокол работы;
* при неуспешном распознавании выдается сообщение об ошибке.

**3.10 Контрольный пример**

Результат работы лексического анализатора, полученный при выполнении контрольного примера представлен в приложении В.

Глава 4 Разработка синтаксического анализатора

4.1 Структура синтаксического анализатора

Синтаксический анализ – это фаза трансляции, выполняемая после лексического анализа и предназначенная для распознавания синтаксических конструкций. Структура синтаксического анализатора представлена на рисунке 4.1.

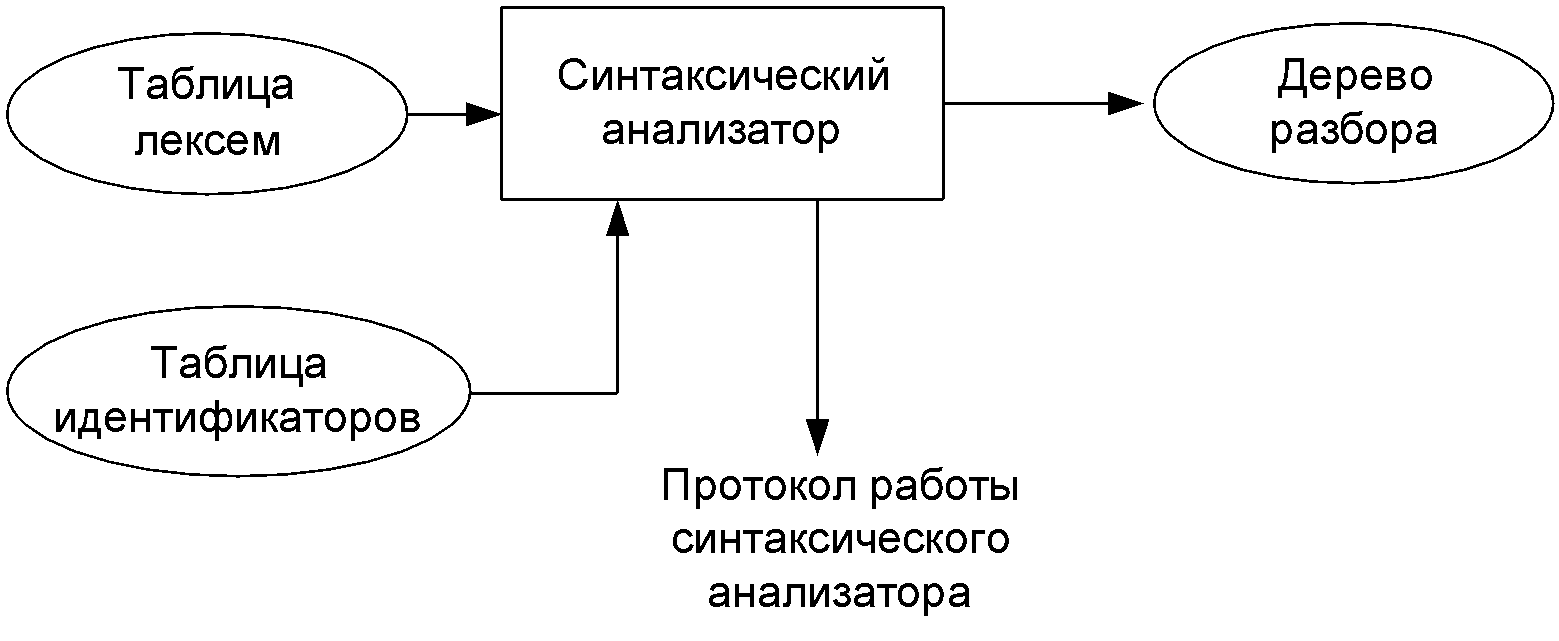


Рисунок 4.1 – Структура синтаксического анализатора

Входом для синтаксического анализа является таблица лексем и таблица идентификаторов, полученные после фазы лексического анализа. Выход – дерево разбора.

4.2 Контекстно-свободная грамматика, описывающая синтаксис языка

В синтаксическом анализаторе транслятора языка KVV-2024 используется контекстно-свободная грамматика , где:

* T – множество терминальных символов;
* N – множество нетерминальных символов (первый столбец таблицы 4.1);
* P – множество правил языка (второй столбец таблицы 4.1);
* S – начальный символ грамматики, являющийся нетерминалом.

Эта грамматика имеет нормальную форму Грейбах, т.к. она не леворекурсивная (не содержит леворекурсивных правил) и правила  имеют вид:

1. , где ; (или , или )
2. , где — начальный символ, при этом если такое правило существует, то нетерминал  не встречается в правой части правил.

T – терминальные символы, которыми являются сепараторы, знаки арифметических операций и некоторые строчные буквы, N – нетерминальные символы, представленные заглавными буквами латинского алфавита..

Грамматика языка KVV-2024 представлена в приложении Г.

Таблица 4.1 – Перечень правил, составляющих грамматику языка и описание нетерминальных символов KVV-2024

|  |  |  |
| --- | --- | --- |
| Нетерминал | Цепочки правил | Описание |
| S | fti(F){NeE;};S  m{NeE;}; | Правила для общих структур |
| N | dti;  dti;N  oE;  oE;N  i=E;N  i=E;  c(W)[N]N | Правила для конструкций и инструкций |
| E | i  l  I  i(W)  i()M  iM  lM  IM  I(W)M | Правила для выражений |
| M | vE  vEM | Правила для операторов |
| F | ti  ti,F | Правила для параметров функции |
| W | i  l  I,W  l,W | Правила для параметров вызываемой функции |

Грамматика Грейбаха — это форма контекстно-свободной грамматики, в которой каждая продукция (правило вывода) имеет строго определенный вид.

4.3 Построение конечного магазинного автомата

Конечный магазинный автомат (КМА) — это вычислительная модель, которая расширяет возможности конечного автомата, добавляя стековую память (или «магазин»). Это позволяет автомату работать с контекстно-свободными языками, что делает его мощнее, чем стандартный конечный автомат,

В данном курсовом проекте грамматика приведена к нормальной форме Грейбах. Это означает, что каждое правило имеет вид A → aα, где a∈T, α∈N. Конечный автомат с магазинной памятью можно представить в виде семёрки .Подробное описание компонентов магазинного автомата представлено в таблице 4.2.

Таблица 4.2 – Описание компонентов магазинного автомата

|  |  |  |
| --- | --- | --- |
| Компонента | Определение | Описание |
|  | Множество состояний автомата | Состояние автомата представляет из себя структуру, содержащую позицию на входной ленте, номера текущего правила и цепочки и стек автомата |
|  | Алфавит входных символов | Алфавит представляет из себя множества терминальных и нетерминальных символов, описание которых содержится в таблица 3.1 и 4.1. |
|  | Алфавит специальных магазинных символов | Алфавит магазинных символов содержит стартовый символ и маркер дна стека (представляет из себя символ $) |
|  | Функция переходов автомата | Функция представляет из себя множество правил грамматики, описанных в таблице 4.1. |
|  | Начальное состояние автомата | Состояние, которое приобретает автомат в начале своей работы. Представляется в виде стартового правила грамматики |
|  | Начальное состояние магазина автомата | Символ маркера дна стека $ |

Из данной таблицы можно сделать вывод, что магазинный автомат используется для анализа и обработки языка KVV-2024 с использованием контекстно-свободной грамматики. Автомат состоит из состояний, алфавитов символов, функции переходов и имеет начальное и конечные состояния.

4.4 Основные структуры данных

Основные структуры данных синтаксического анализатора включают в себя структуру магазинного автомата и структуру грамматики Грейбах, описывающей правила языка KVV-2024. Данные структуры представлены в приложении Д.

4.5 Описание алгоритма синтаксического разбора

Принцип работы автомата следующий:

1. В магазин записывается стартовый символ.
2. На основе полученных ранее таблиц формируется входная лента.
3. Запускается автомат.
4. Выбирается цепочка, соответствующая нетерминальному символу, записывается в магазин в обратном порядке.
5. Если терминалы в стеке и в ленте совпадают, то данный терминал удаляется из ленты и стека. Иначе возвращаемся в предыдущее сохраненное состояние и выбираем другую цепочку нетерминала.
6. Если в магазине встретился нетерминал, переходим к пункту 4.
7. Если наш символ достиг дна стека, и лента в этот момент пуста, то синтаксический анализ выполнен успешно. После 3 исключений синтаксический анализатор завершает работу и генерирует последнее исключение.
   1. **Структура и перечень сообщений синтаксического анализатора**

Перечень сообщений синтаксического анализатора представлен в таблице 4.3.

Таблица 4.3 – Перечень сообщений синтаксического анализатора

|  |  |
| --- | --- |
| Код | Сообщение |
| 600 | Ошибка синтаксического анализа: неверная структура программы |
| 601 | Ошибка синтаксического анализа: ошибочный оператор |
| 602 | Ошибка синтаксического анализа: ошибка в выражении |
| 603 | Ошибка синтаксического анализа: ошибка в параметрах функции |
| 604 | Ошибка синтаксического анализа: ошибка в параметрах вызываемой функции |
| 605 | Ошибка синтаксического анализа: ошибка в подвыражении |
| 609 | Ошибка синтаксического анализа: неверный номер правила |

Таким образом, при написании кода, достаточно просто понять, с чем конкретно связана та или иная ошибка.

**4.7 Параметры синтаксического анализатора**

Входным параметром синтаксического анализатора является таблица лексем, полученная на этапе лексического анализа, а также правила контекстно-свободной грамматики в форме Грейбах.

Выходными параметрами являются трассировка прохода таблицы лексем и правила разбора, которые записываются в файл протокола данного этапа обработки.

## **4.8 Принцип обработки ошибок**

Обработка ошибок происходит следующим образом:

1. Синтаксический анализатор перебирает все правила и цепочки правила грамматики для нахождения подходящего соответствия с конструкцией, представленной в таблице лексем.
2. Если невозможно подобрать подходящую цепочку, то генерируется соответствующая ошибка, которая записывается в протокол работы и работа транслятора останавливается.

4.9 Контрольный пример

Результатом работы синтаксического анализатора является дерево разбора. Результат работы синтаксического анализатора представлен в приложении Е.

Глава 5 Разработка семантического анализатора

5.1 Структура семантического анализатора

Семантический анализатор принимает на свой вход результаты работ лексического и синтаксического анализаторов, то есть таблицы лексем, идентификаторов и результат работы синтаксического анализатора, то есть дерево разбора, и последовательно ищет необходимые ошибки. Некоторые проверки (такие как проверка на единственность точки входа, проверка на предварительное объявление переменной) осуществляются в процессе лексического анализа. Общая структура обособленно работающего (не параллельно с лексическим анализом) семантического анализатора представлена на рисунке 5.1.



Рисунок 5.1. Структура семантического анализатора

Семантический анализатор состоит из набора функций для проверки правильности исходной программы. Часть ошибок семантического анализа обрабатываются на этапе лексического анализа. Но ошибки, требующие более сложной обработки (например, несоответствие типов операндов) вынесены в отдельный этап, следующий после синтаксического анализа.

5.2 Функции семантического анализатора

Семантический анализатор выполняет проверку на основные правила языка (семантики языка), которые описаны в разделе 1.16.

5.3 Структура и перечень сообщений семантического анализатора

Сообщения, формируемые семантическим анализатором, представлены в таблице 5.1.

Таблица 5.1 – Перечень сообщений семантического анализатора

|  |  |
| --- | --- |
| Код | Сообщение |
| 700 | Ошибка семантического анализа: в функции отсутствует return |
| 701 | Ошибка семантического анализа: функция не всегда возвращает значение |

Окончание таблицы 5.1

|  |  |
| --- | --- |
| 702 | Ошибка семантического анализа: попытка объявить функцию в main |
| 703 | Ошибка семантического анализа: ошибка в блоке main: функция возвращает не целочисленное занчение |
| 704 | Ошибка семантического анализа: тип возвращаемого значения не соответствует типу функции |
| 705 | Ошибка семантического анализа: слишком много аргументов передано в функцию |
| 706 | Ошибка семантического анализа: тип передаваемого аргумента не соответствует типу параметра функции |
| 707 | Ошибка семантического анализа: слишком мало аргументов передано в функцию |
| 708 | Ошибка семантического анализа: попытка переопределить библиотечную функцию |
| 709 | Ошибка семантического анализа: неопознаная библиотечная функция |
| 710 | Ошибка семантического анализа: в выражении используется не соответствующий тип данных |
| 711 | Ошибка семантического анализа: в выражении используется не соответствующая ,по типу данных, функция |

Таким образом, при написании кода, достаточно просто понять, с чем конкретно связана та или иная ошибка.

5.4 Принцип обработки ошибок

В качестве входных параметров выступают таблица литералов и идентификаторов. Далее происходит полный анализ данных таблиц. В случае возникновения ошибок, вызываем функцию получения ошибки, которая принимает обязательным параметром код ошибки в таблице сообщений, номер строки и позицию. Пользователь может ознакомиться с данной ошибкой, открыв протокол.

5.5 Контрольный пример

Результат работы семантических функций описан в главе 8.

Глава 6 Вычисление выражений

6.1 Выражения, допускаемые языком

В языке KVV-2024 допускаются вычисления выражений целочисленного типа данных с поддержкой вызова функций внутри выражений. Приоритет операций представлен на таблице 6.1.

Таблица 6.1. Приоритеты операций

|  |  |
| --- | --- |
| Операция | Значение приоритета |
| ( ) | 3 |
| \* | 2 |
| / | 2 |
| + | 1 |
| - | 1 |

Выражения и операции, допускаемые языком, подробно описаны в разделе 1.12 и 1.13.

6.2 Польская запись и принцип ее построения

Польская запись – это альтернативный способ записи арифметических выражений, преимущество которого состоит в отсутствии скобок, а также намного более простой обработки выражений впоследствии.

Обратная польская запись – это форма записи математических и логических выражений, в которой операнды расположены перед знаками операций.

Алгоритм построения:

* читаем очередной символ;
* если он является идентификатором или литералом, то добавляем его выходной строке;
* если символ является символом функции, то помещаем его в стек;
* если символ является открывающей скобкой, то она помещается в стек;
* исходная строка просматривается слева направо;
* если символ является закрывающей скобкой, то выталкиваем из стека в выходную строку все символы пока не встретим открывающую скобку. При этом обе скобки удаляются и не попадают в выходную строку;
* как только входная лента закончится все символы из стека выталкиваются в выходную строку;
* в случае если встречаются операции, то выталкиваем из стека в выходную строку все операции, которые имеют выше приоритетность чем последняя операция;
* также, если идентификатор является именем функции, то он заменяется на спецсимвол «@».

Пример преобразования выражения в обратную польскую запись, представлен в таблице 6.1.

Таблица 6.1 – Пример преобразования выражения в обратную польскую запись

|  |  |  |
| --- | --- | --- |
| Выражение | Стек | Результат |
| (x + y)\*7 |  |  |
| x + y)\*7 | ( |  |
| + y)\*7 | ( | x |
| y)\*7 | ( + | x |
| )\*7 | ( + | x y |
| \*7 |  | x y + |
| 7 | \* | x y + |
|  | \* | x y + 7 |
|  |  | x y + 7 \* |

Таким образом, в результате преобразования все выражения языка KVV-2024 представляются в виде обратной польской записи, что позволяет выполнять их вычисление без использования скобок и с легкостью определять порядок операций.

6.3 Программная реализация обработки выражений

Программная реализация алгоритма преобразования выражений к польской записи представлена в приложении Ж.

6.4 Контрольный пример

Пример преобразования выражения к польской записи представлен в таблице 6.2. Преобразование выражений в формат польской записи необходимо для построения более простых алгоритмов их вычисления.

ПОЛИЗ, или, как её еще называют, обратная польская запись, это способ бесско­боч­ного представления выражений (не только арифметических), в которых операнды пред­шествуют операции.

Таблица 6.2. Приведение выражений к ПОЛИЗ

|  |  |
| --- | --- |
| Выражение | Обратная польская запись выражения |
| (i+i) | ii+ |
| R(l)\*P(l,l) | l@ll,P\* |
| i%l+l | il%l+ |

Таким образом, исходя из вышеприведённой таблицы, преобразование выражений в обратную польскую запись в языке KVV-2024 упрощает алгоритмы их вычисления и преобразования к ассемблерному коду.

Глава 7 Генерация кода

7.1 Структура генератора кода

Генерация кода – заключительный этап работы транслятора. Результатом данного этапа будет код, сгенерированный для выполнения на ассемблере на основе таблицы лексем и таблицы идентификаторов, что является требуемым результатом работы программы.Транслятор кода начинает свою работу только в том случае если код на языке KVV-2024 прошёл предыдущие компоненты транслятора без ошибок. Схема данного этапа изображена на рисунке 7.1.

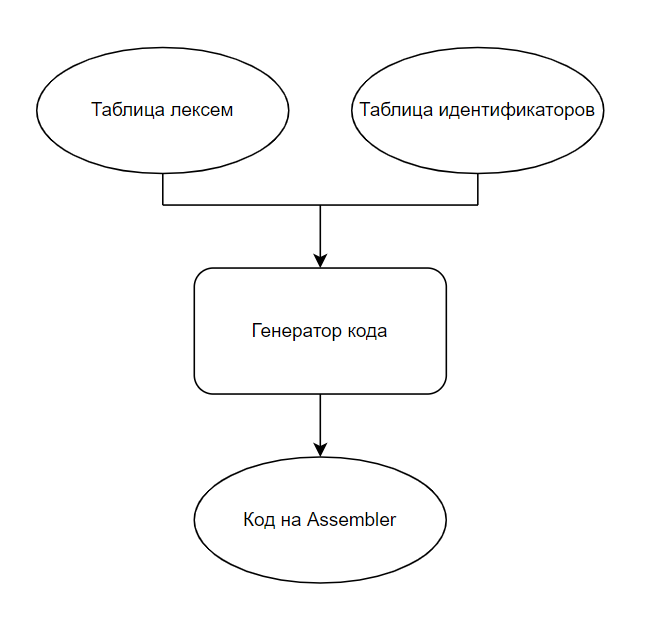


Рисунок 7.1 – Структура генератора кода

Таким образом, генератор кода выполняет не менее значимую часть компиляции.

7.2 Представление типов данных в оперативной памяти

Элементы таблицы идентификаторов расположены в разных сегментах языка ассемблера – .data и .const. Идентификаторы языка KVV-2024 размещены в сегменте данных(.data). Литералы – в сегменте констант (.const). Соответствия между типами данных идентификаторов на языке KVV-2024 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка KVV-2024 и языка ассемблера

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке KVV-2024 | Тип идентификатора на языке ассемблера | Пояснение |
| uint | DWORD | Хранит целочисленный тип данных без знака (4 байта). |
| string | DWORD | Хранит строку (1 байт). |

Следовательно, таблица 7.1 показывает соответствия между типами идентификаторов на языке KVV-2024 и языке ассемблера. Это важно при переводе кода с языка KVV-2024 на язык ассемблера, чтобы правильно определить типы данных и использовать соответствующие инструкции и регистры для работы с идентификаторами.

7.3 Статическая библиотека

В языке KVV-2024 предусмотрена статическая библиотека. Статическая библиотека содержит функции, написанные на языке C++. Объявление функций статической библиотеки генерируется автоматически в коде ассемблера. Объявление функций статической библиотеки генерируется автоматически.

Таблица 7.2 – Функции статической библиотеки

|  |  |  |
| --- | --- | --- |
| Функция(C++) | Возвращаемое значение | Описание |
| unsigned int toPow(unsigned int a, unsigned int b) | uint | Функция возведения числа в степень |
| unsigned int random(unsigned int a) | uint | Функция вывода на консоль строкового идентификатора/литерала |
| int writenum(unsigned int value) | uint | Функция выводит переданное целое число value на экран без перевода строки. |
| int writestr(char\* ptr) | uint | Функция выводит строку, переданную через указатель ptr. Если указатель ptr равен nullptr, выводит только перевод строки. |
| unsigned int writenumline(unsigned int value) | uint | Функция выводит целое число value на экран, добавляя перевод строки. |
| unsigned int writestrline(char\* ptr) | uint | Функция выводит строку, переданную через указатель ptr, с добавлением перевода строки. |

Объявление функций статической библиотеки генерируется автоматически.

7.4 Особенности алгоритма генерации кода

В языке KVV-2024 генерация кода строится на основе таблиц лексем и идентификаторов. Общая схема работы генератора кода представлена на рисунке 7.2.

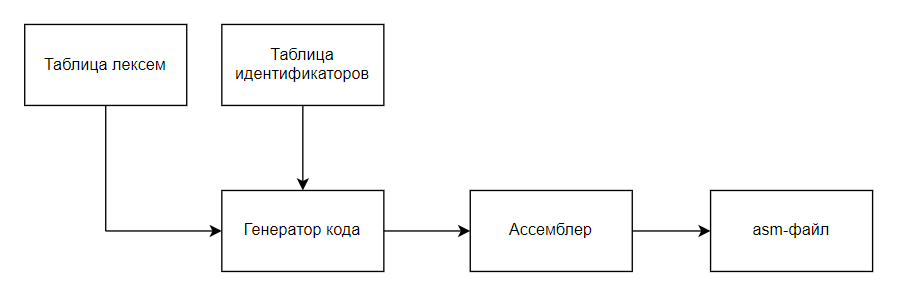


Рисунок 7.2 – Структура генератора кода

Алгоритмы генерации кода варьируются в зависимости от конкретной задачи и языка программирования.

7.5 Входные параметры, управляющие генерацией кода

На вход генератора кода поступают таблицы лексем и идентификаторов исходного кода программы на языке KVV-2024. Результаты работы генератора кода выводятся в файл с расширением \*.asm.

**7.6 Контрольный пример**

Результат генерации ассемблерного кода на основе контрольного примера из приложения А приведен в приложении З. Результат работы контрольного примера приведён на рисунке 7.3.

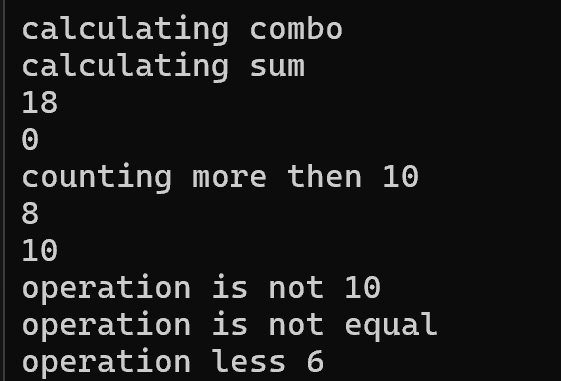


Рисунок 7.3 – Результат работы программы на языке KVV-2024

**Генерация ассемблерного кода** — это этап трансляции программы на высокоуровневом языке (например, C++) в **низкоуровневый ассемблерный код**, который понятен процессору. Этот этап является частью работы транслятора, и результатом его работы становится ассемблерный код, который затем может быть преобразован в машинные команды.

Глава 8 Тестирование транслятора

8.1 Общие положения

Тестирование должно покрывать как можно больше сценариев использования языка и его конструкций. Все тесты были представлены для типичных ошибок пользователей при использовании языка. Когда компилятор обнаруживает ошибку, он записывает информацию о ней в протокол, содержащий номер ошибки и диагностическое сообщение, помогающее разработчику понять причину ошибки компиляции. Результаты тестирования записываются в файл .log.

8.2 Результаты тестирования

В языке KVV-2024 не разрешается использовать запрещённые входным алфавитом символы. Результат использования запрещённого символа показан в таблице 8.1.

Таблица 8.1 – Тестирование фазы проверки на допустимость символов

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| message = 'Б'; | 'Б' - (129)Ошибка лексического анализа: неопознанная лексема в строке 36  Лексический анализ выполнен c ошибками |
| message = 'DDD'; | 'DDD' - (129)Ошибка лексического анализа: неопознанная лексема в строке 36  Лексический анализ выполнен c ошибками |

На этапе лексического анализа могут возникнуть ошибка, описанная в пункте 3.6. Результаты тестирование лексического анализатора показано в таблице 8.2.

Таблица 8.2 – Тестирование лексического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| operation = random(5) \* pow(333333333333333333333333333333333333333333333333333333333333333333 ,2); | Ошибка 132: лексического анализа: значение литерала превышает максимальное значение типа uint |
| declare uint res = multisum(4,5); | Ошибка124: лексического анализа: идентификатор не объявлен |
| main  {  };  main  {  }; | Ошибка 131: Несколько точек входа |

На этапе синтаксического анализа могут возникнуть ошибки, описанные в пункте 4.6. Результаты тестирования синтаксического анализатора показаны в таблице 8.3.

Таблица 8.3 – Тестирование синтаксического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| declare string x; | Ошибка 600: Неверная структура программы. |

Итоги тестирования семантического анализатора приведены в таблице 8.4.

Таблица 8.4 – Тестирование семантического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| declare uint x;  x = ‘f’; | Ошибка 705: Несоответствие присваиваемого типа данных. |
| main  {  return ‘s’;  }; | Ошибка 703: Ошибка семантического анализа: ошибка в блоке main: функция возвращает не целочисленное занчение |
| uint function powe(uint n, uint m)  {  write('calculating sum');  }; | Ошибка 700: Ошибка семантического анализа: в функции отсутствует return |

Таким образом данный раздел предоставляет набор тестов для проверки лексического, синтаксического и семантического анализаторов.

# **Глава 9 Разработка и тестирование интерпретатора**

## **9.1 Структура и перечень сообщений времени выполнения**

Разработка интерпретатора — это процесс создания программного обеспечения, которое выполняет анализ, интерпретацию и выполнение исходного кода или промежуточного представления программы.

Интерпретатор — это программа, которая выполняет код, преобразовывая его в машинные команды во время выполнения, без предварительного этапа компиляции в исполняемый файл.

Сообщения времени выполнения информируют пользователя об ошибках, предупреждениях или информационных событиях, возникающих во время выполнения программы. Пример сообщений времени выполнений представлен на рисунке 9.1.

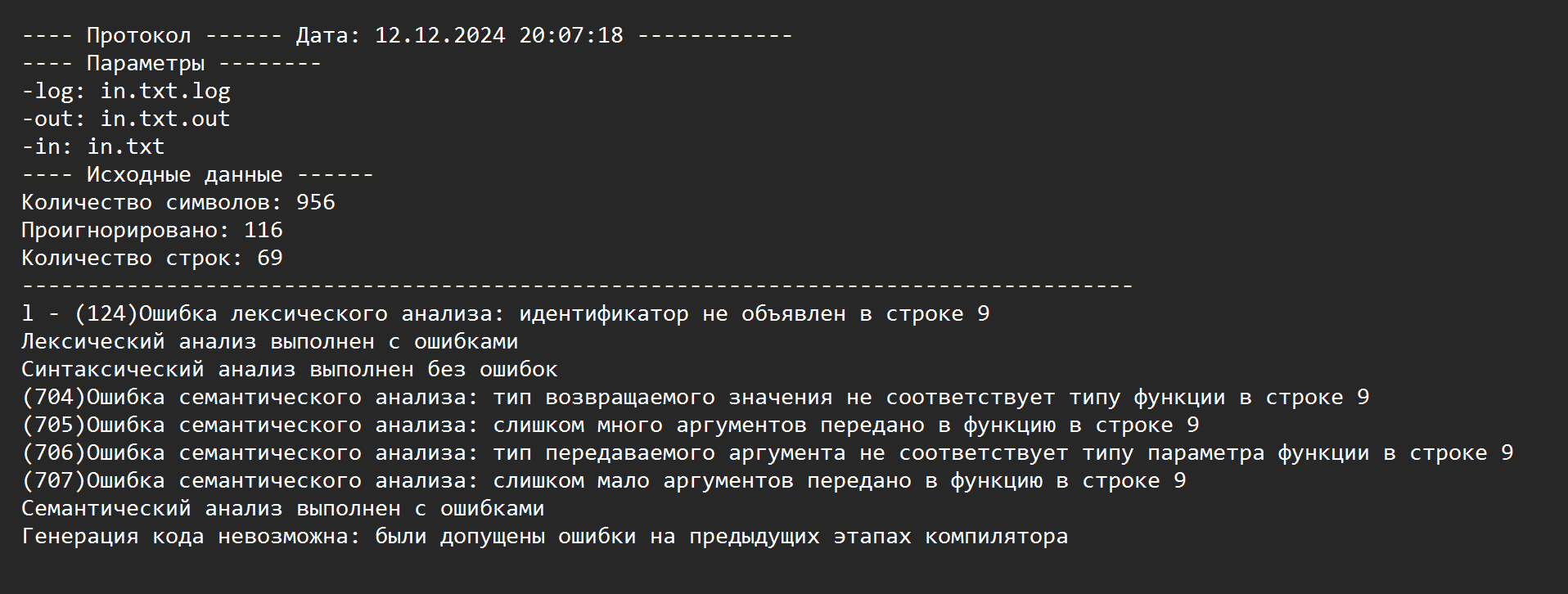


Рисунок 9.1 Пример сообщений времени выполнений

Сообщения времени выполнения, такие как на представленном рисунке, представляют собой четкую и структурированную диагностику работы интерпретатора. Они помогают пользователю понять, на каком этапе анализа произошла ошибка, и предоставляют подробное описание для её устранения. Такая структура способствует улучшению разработки, сокращению времени на исправление ошибок и упрощает работу с программным кодом.

# **Заключение**

В данном курсовом проекте были выполнены поставленные минимальные требования. В ходе работы было изучено много нового, а также закреплены знания, которые были получены ранее. Также стоит отметить, что данный курсовой проект позволил совместить закрепление знаний сразу по двум языкам программирования, таких как C++ и Ассемблер. При написании приложения были усвоены такие понятия как синтаксический, лексический и семантический анализатор и т.п. В итоге был получен примитивный язык программирования KVV-2024, который не имеет сложных конструкций, которые реализованы на сегодняшний день во многих других языках программирования.

Окончательная версия языка KVV-2024 включает:

1. 2 типа данных;
2. Поддержка оператора вывода;
3. Возможность подключения и вызова функций стандартной библиотеки;
4. Наличие 5 арифметических и 6 операторов сравнения для вычисления выражений;
5. Возможность вызова функции в выражении.

# **Список использованных источников**

1. Ахо, А. Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Дж. Ульман. – M.: Вильямс, 2003. – 768с.

2. Ахо, А. Теория синтаксического анализа, перевода и компиляции /А. Ахо, Дж. Ульман. – Москва : Мир, 1998. – Т. 2 : Компиляция. - 487 с.

3. Герберт, Ш. Справочник программиста по C/C++ / Шилдт Герберт. - 3-е изд. – Москва : Вильямс, 2003. - 429 с.

4. Прата, С. Язык программирования С++. Лекции и упражнения / С. Прата. – М., 2006 — 1104 c.

5. Страуструп, Б. Принципы и практика использования C++ / Б. Страуструп – 2009 – 1238 с

# **Приложение А**

uint function countsum(uint n, uint m)

{

write('calculating sum');

return (n+m);

};

uint function coutsomethingelse(uint n, uint m)

{

write('calculating combo');

return (n + m + countsum(n,m));

};

main

{

declare uint operation = coutsomethingelse(4,5);

write(operation);

declare uint hexvalue = 0x0;

write(hexvalue);

if(operation » 10)

[

write('counting more then 10');

]

else

[

write('counting less then 10');

write('l');

];

declare uint counter;

counter = 0;

operation = random(5) \* pow(3 ,2);

operation = (operation-1) / 3;

write(operation);

declare string message;

message = 'new calculating';

operation = operation % 7;

if(10 ~ 10)

[

write('10 is 10');

];

if(operation > 0)

[

write('operation is not 10');

];

Окончание приложения А

if(operation ! 5)

[

write('operation is not equal');

];

if(operation < 6)

[

write('operation less 6');

];

return 0;

};

Листинг 1 - Исходный код на языке KVV-2024

# **Приложение Б**

|  |
| --- |
| namespace LT {  struct Entry {  char lexemе;  int sn;  int idxTI;  };  struct LexTable  {  int maxsize;  int size;  Entry\* table;  };  LexTable Create(int size);  void Add(LexTable& lextable, Entry entry);  Entry GetEntry(LexTable& lextable, int n);  void Delete(LexTable& lextable); |

Листинг 2 – Структура таблицы лексем

|  |
| --- |
| namespace IT {  enum IDDATATYPE { UINT = 1,STR = 2};//типы данных идентификаторов: числовой, строковый  enum IDTYPE { V = 1, F = 2, P = 3, L = 4, S = 5, Z = 6 };//типы идентификаторов: V = переменная, F = функция, P =параметр, L =литерал, S = стандартная функция Z - знак  static IDDATATYPE LENGHT\_PARAMS[] = { IT::STR };  static IDDATATYPE LENGHT\_POW[] = {IT::UINT,IT::UINT };  static IDDATATYPE LENGHT\_RAND[] = {IT::UINT };  struct Entry {  int idxfirstLE;  char scope[SCOPE\_MAXSIZE];  char id[ID\_MAXSIZE];  IDDATATYPE iddatatype;  IDTYPE idtype;  struct parm  {  IDDATATYPE\* types;  short amount;  parm(int amount, IDDATATYPE\* types)  {  this->amount = amount;  this->types = types;  }  };  union {  unsigned int vint;  struct {  char len;  char str[TI\_STR\_MAXSIZE - 1];  } vstr;  double vdouble;  struct  {  IDDATATYPE\* types;    short amount;  } params;  } value;  Entry() = default;  Entry(int idxfirstLE, const char\* scope, const char\* id, IDTYPE idtype, int value);  Entry(int idxfirstLE, const char\* scope, const char\* id, IDTYPE idtype, double value);  Entry(int idxfirstLE, const char\* scope, const char\* id, IDTYPE idtype, const char\* value);  Entry(int idxfirstLE, const char\* scope, const char\* id, IDDATATYPE iddatatype, IDTYPE idtype);  Entry(int idxfirstLE, const char\* scope, const char\* id, IDDATATYPE iddatatype, IDTYPE idtype, Entry::parm value);  }; |

Окончание приложения Б - стурктуры таблицы идентификаторов

|  |
| --- |
| struct IdTable {  int maxsize;  int size;  Entry\* table;  };  IdTable Create(int size);  void Add(IdTable& idtable, const Entry& entry);  Entry GetEntry(IdTable& idtable, int n);  int IsId(IdTable& idtable, const char scope[SCOPE\_MAXSIZE], const char id[ID\_MAXSIZE]);  int IsId(IdTable& idtable, const char scope[SCOPE\_MAXSIZE], const char id[ID\_MAXSIZE], bool lexFlag);  int IsLiteral(IdTable& idtable, const char literal[TI\_STR\_MAXSIZE]);  int IsLiteral(IdTable& idtable, int literal);  void Delete(IdTable& idtable);  } |

Листинг 3 – Структура таблицы идентификаторов

**Приложение В**

|  |
| --- |
| 01 tfi(ti,ti)  02 {  03 p(l);  04 r(i+i);  05 };  06 tfi(ti,ti)  07 {  08 p(l);  09 r(i+i+i(i,i));  10 };  11 m  12 {  13 dti=i(l,l);  14  15 p(i);  16  17 dti=l;  18 p(i);  19 u(i`l)  20 [  21 p(l);  22 ]  23 E  24 [  25 p(l);  26 p(l);  27 ];  28  29 dti;  30 i=l;  31  32 i=R(l)\*P(l,l);  33 i=(i-l)/l;  34 p(i);  35 dti;  36 i=l;  37 i=i%l;  38 u(l~l)  39 [  40 p(l);  41 ];  42 u(l!l)  43 [  44 p(l);  45 ];  46 u(i<l)  47 [  48 p(i);  49 ];  50 u(i>l)  51 [  52 p(i);  53 ];  54 rl;  55 }; |

Листинг 4 – Результат лексического анализа

# **Приложение Г**

|  |
| --- |
| namespace GRB {  #pragma region Rules  const Greibach greibach(NS('S'), TS('$'),  9,  Rule(NS('S'), GRB\_ERROR\_SERIES + 0,//начало кода s-начальный символ(следущее функцию) N-выпажение r-ретурн E-литералы t-тип f-функций  7,  Rule::Chain(8, TS('m'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';')),  Rule::Chain(7, TS('m'), TS('{'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';')),  Rule::Chain(9, TS('m'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S')),  Rule::Chain(14, TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S')),  Rule::Chain(13, TS('t'), TS('f'), TS('i'), TS('('), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S')),    Rule::Chain(13, TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';')),  Rule::Chain(12, TS('t'), TS('f'), TS('i'), TS('('), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'))  ),  Rule(NS('N'), GRB\_ERROR\_SERIES + 1,//выражения d-declare(обявление) i-индентификатор l-литерал Q-16с.с p-вывод  24,  Rule::Chain(4, TS('d'), TS('t'), TS('i'), TS(';')),  Rule::Chain(5, TS('d'), TS('t'), TS('i'), TS(';'), NS('N')),  Rule::Chain(6, TS('d'), TS('t'), TS('i'), TS('='), TS('l'), TS(';')),  Rule::Chain(7, TS('d'), TS('t'), TS('i'), TS('='), TS('l'), TS('Q'), TS(';')),  Rule::Chain(6, TS('d'), TS('t'), TS('i'),TS('='),NS('E'), TS(';')),  Rule::Chain(7, TS('d'), TS('t'), TS('i'),TS('='),NS('E'), TS(';'), NS('N')),  Rule::Chain(7, TS('d'), TS('t'), TS('i'), TS('='), TS('l'), TS(';'), NS('N')),  Rule::Chain(8, TS('d'), TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS(';')),  Rule::Chain(9, TS('d'), TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS(';'), NS('N')),  Rule::Chain(3, TS('r'), NS('E'), TS(';')),  Rule::Chain(4, TS('i'), TS('='), NS('E'), TS(';')),  Rule::Chain(5, TS('i'), TS('='), NS('E'), TS(';'), NS('N')),  Rule::Chain(4, TS('r'), NS('E'), TS(';'), NS('N')),  Rule::Chain(4, TS('p'), NS('E'), TS(';'), NS('N')),  Rule::Chain(3, TS('p'), NS('E'), TS(';')),      Rule::Chain(3, TS('p'), TS('i'), TS(';')),    Rule::Chain(10, TS('u'), TS('('), NS('V'), NS('C'), NS('V'),TS(')'), TS('['), NS('N'),TS(']'), TS(';') ),  Rule::Chain(14, TS('u'), TS('('), NS('V'), NS('C'), NS('V'),TS(')'), TS('['), NS('N'),TS(']'), TS('E'), TS('['), NS('N'), TS(']'), TS(';') ),  Rule::Chain(15, TS('u'), TS('('), NS('V'), NS('C'), NS('V'),TS(')'), TS('['), NS('N'),TS(']'), TS('E'), TS('['), NS('N'), TS(']'), TS(';'), NS('N') ),  Rule::Chain(11, TS('u'), TS('('), NS('V'), NS('C'), NS('V'),TS(')'), TS('['), NS('N'),TS(']'), TS(';'), NS('N') ),  Rule::Chain(11, TS('W'), TS('('), NS('V'), NS('C'), NS('V'),TS(')'), TS('['), NS('N'),TS(']'), TS(';'), NS('N') ),  Rule::Chain(10, TS('W'), TS('('),NS('V'), NS('С'), NS('V'),TS(')'), TS('['), NS('N'),TS(']'), TS(';') ),  Rule::Chain(3, TS('p'), TS('l'), TS(';')),  Rule::Chain(6, TS('p'), TS('i'), TS('('), NS('W'), TS(')'), TS(';'))  ),  Rule(NS('E'), GRB\_ERROR\_SERIES + 2,// литералы u-названеи функции V-опернд истинности C- L P-степень R-рандом M-операции  24,  Rule::Chain(1, TS('i')),  Rule::Chain(4, TS('L'),TS('('), TS('l'), TS(')')),  Rule::Chain(4, TS('L'),TS('('), TS('i'), TS(')')),  Rule::Chain(5, TS('L'),TS('('), TS('i'), TS(')'), NS('M')),    Rule::Chain(6, TS('P'),TS('('), NS('V'), TS(','), NS('V'), TS(')')),  Rule::Chain(7, TS('P'),TS('('), NS('V'), TS(','), NS('V'), TS(')'), NS('M')),    Rule::Chain(4, TS('R'),TS('('), TS('l'), TS(')')),  Rule::Chain(4, TS('R'),TS('('), TS('i'), TS(')')),  Rule::Chain(5, TS('R'),TS('('), TS('l'), TS(')'), NS('M')),  Rule::Chain(5, TS('R'),TS('('), TS('i'), TS(')'), NS('M')),  Rule::Chain(4, TS('N'),TS('('), TS('l'), TS(')')),  Rule::Chain(5, TS('N'),TS('('), TS('l'), TS(')'), NS('M')),  Rule::Chain(6, TS('C'),TS('('), TS('l'), TS(','), TS('l'), TS(')')),  Rule::Chain(5, TS('L'),TS('('), TS('l'), TS(')'), NS('M')),    Rule::Chain(1, TS('l')),  Rule::Chain(2, TS('l'),NS('M')),  Rule::Chain(3, TS('('), NS('E'), TS(')')),  Rule::Chain(4, TS('i'), TS('('), NS('W'), TS(')')),  Rule::Chain(2, TS('i'), NS('M')),    Rule::Chain(2, TS('l'), NS('M')),  Rule::Chain(3,TS('i'),TS('('),TS(')')),  Rule::Chain(4,TS('i'),TS('('),TS(')'),NS('M')),  Rule::Chain(4, TS('('), NS('E'), TS(')'), NS('M')),  Rule::Chain(5, TS('i'), TS('('), NS('W'), TS(')'), NS('M'))    ),  Rule(NS('M'), GRB\_ERROR\_SERIES + 3,//операции  20,  Rule::Chain(2, TS('+'), NS('E')),  Rule::Chain(4, TS('+'), TS('('), NS('E'), TS(')')),  Rule::Chain(5, TS('+'), TS('('), NS('E'), TS(')'), NS('M')),  Rule::Chain(3, TS('+'), NS('E'), NS('M')),    Rule::Chain(2, TS('-'), NS('E')),  Rule::Chain(4, TS('-'), TS('('), NS('E'), TS(')')),  Rule::Chain(5, TS('-'), TS('('), NS('E'), TS(')'), NS('M')),  Rule::Chain(3, TS('-'), NS('E'), NS('M')),  Rule::Chain(2, TS('\*'), NS('E')),  Rule::Chain(4, TS('\*'), TS('('), NS('E'), TS(')')),    Rule::Chain(5, TS('\*'), TS('('), NS('E'), TS(')'), NS('M')),  Rule::Chain(3, TS('\*'), NS('E'), NS('M')),  Rule::Chain(2, TS('/'), NS('E')),  Rule::Chain(4, TS('/'), TS('('), NS('E'), TS(')')),  Rule::Chain(5, TS('/'), TS('('), NS('E'), TS(')'), NS('M')),    Rule::Chain(3, TS('/'), NS('E'), NS('M')),  Rule::Chain(2, TS('%'), NS('E')),  Rule::Chain(4, TS('%'), TS('('), NS('E'), TS(')')),  Rule::Chain(5, TS('%'), TS('('), NS('E'), TS(')'), NS('M')),  Rule::Chain(3, TS('%'), NS('E'), NS('M'))    ),  Rule(NS('F'), GRB\_ERROR\_SERIES + 4,//параметры функции  2,  Rule::Chain(2, TS('t'), TS('i')),  Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('F'))  ),  Rule(NS('W'), GRB\_ERROR\_SERIES + 5,//аргументы  4,  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l')),    Rule::Chain(3, TS('i'), TS(','), NS('W')),  Rule::Chain(3, TS('l'), TS(','), NS('W'))    ),  Rule(NS('C'), GRB\_ERROR\_SERIES + 6,//операторы истинности  6,    Rule::Chain(1, TS('>')),  Rule::Chain(1, TS('<')),    Rule::Chain(1, TS('!')),    Rule::Chain(1, TS('~')),    Rule::Chain(1, TS('$')),    Rule::Chain(1, TS('`'))  ),  Rule(NS('V'), GRB\_ERROR\_SERIES + 7,//операнды истинности  2,  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l'))  ),  Rule(NS('Q'), GRB\_ERROR\_SERIES + 8,  2,  Rule::Chain(1, TS('i')),  Rule::Chain(2, TS('i'), TS('Q'))  )  ); |

Листинг 5 – Грамматика языка KVV-2024

# **Приложение Д**

|  |
| --- |
| #define NS(n) Rule::Chain::N(n)  #define TS(n) Rule::Chain::T(n)  //не терменал и терменал  namespace GRB {  struct Rule {  GRBALPHABET nn;  int iderror;  short size;  struct Chain {  short size;  GRBALPHABET\* nt;  Chain();  Chain(short size, GRBALPHABET s, ...);  std::string getCChain();  static GRBALPHABET T(char t) { return GRBALPHABET(t); }  static GRBALPHABET N(char n) { return -GRBALPHABET(n); }  static bool isT(GRBALPHABET s) { return s > 0; }  static bool isN(GRBALPHABET s) { return !isT(s); }  static char alphabet\_to\_char(GRBALPHABET s) { return isT(s) ? char(s) : char(-s); }  } \*chains;  Rule();  Rule(GRBALPHABET nn, int iderror, short size, Chain c, ...);  std::string getCRule(short nchain);  short getNextChain(GRBALPHABET t, Rule::Chain& chain, short n);  };  struct Greibach {  short size;  GRBALPHABET startN;  GRBALPHABET stbottomT;  Rule\* rules;  Greibach();  Greibach(GRBALPHABET startN, GRBALPHABET stbottomT, short size, Rule r, ...);  short getRule(GRBALPHABET nn, Rule& rule);  Rule getRule(short n);};  const Greibach& getGreibach();} |

Листинг 6 – Структура грамматики Грейбаха

# **Приложение Е**

|  |
| --- |
| 0 : S->tfi(F){NrE;};S  4 : F->ti,F  7 : F->ti  11 : N->pE;  12 : E->(E)  13 : E->l  17 : E->(E)  18 : E->iM  19 : M->+E  20 : E->i  25 : S->tfi(F){NrE;};S  29 : F->ti,F  32 : F->ti  36 : N->pE;  37 : E->(E)  38 : E->l  42 : E->(E)  43 : E->iM  44 : M->+E  45 : E->iM  46 : M->+E  47 : E->i(W)  49 : W->i,W  51 : W->i  57 : S->m{NrE;};  59 : N->dti=E;N  63 : E->i(W)  65 : W->l,W  67 : W->l  70 : N->pE;N  71 : E->(E)  72 : E->i  75 : N->dti=E;N  79 : E->l  81 : N->pE;N  82 : E->(E)  83 : E->i  86 : N->u(VCV)[N]E[N];N  88 : V->i  89 : C->`  90 : V->l  93 : N->pE;  94 : E->(E)  95 : E->l  101 : N->pE;N  102 : E->(E)  103 : E->l  106 : N->pE;  107 : E->(E)  108 : E->l  113 : N->dti;N  117 : N->i=E;N  119 : E->l  121 : N->i=E;N  123 : E->R(l)M  127 : M->\*E  128 : E->P(V,V)  130 : V->l  132 : V->l  135 : N->i=E;N  137 : E->(E)M  138 : E->iM  139 : M->-E  140 : E->l  142 : M->/E  143 : E->l  145 : N->pE;N  146 : E->(E)  147 : E->i  150 : N->dti;N  154 : N->i=E;N  156 : E->l  158 : N->i=E;N  160 : E->iM  161 : M->%E  162 : E->l  164 : N->u(VCV)[N];N  166 : V->l  167 : C->~  168 : V->l  171 : N->pE;  172 : E->(E)  173 : E->l  178 : N->u(VCV)[N];N  180 : V->l  181 : C->!  182 : V->l  185 : N->pE;  186 : E->(E)  187 : E->l  192 : N->u(VCV)[N];N  194 : V->i  195 : C-><  196 : V->l  199 : N->pE;  200 : E->(E)  201 : E->i  206 : N->u(VCV)[N];  208 : V->i  209 : C->>  210 : V->l  213 : N->pE;  214 : E->(E)  215 : E->i  221 : E->l |

Листинг 7 – Результат работы синтаксического анализа

**Приложение Ж**

|  |
| --- |
| (l) => l  (i+i) => ii+  (l) => l  (i+i+i(i,i)) => ii+ii@+  i(l,l) => ll@  (i) => i  l => l  (i) => i  (l) => l  (l) => l  (l) => l  l => l  R(l)\*P(l,l) => l@ll@\*  (i-l)/l => il-l/  (i) => i  l => l  i%l => il%  (l) => l  (l) => l  (i) => i  (i) => i  l => l |

Листинг 8 – Преобразование к польской записи

**Приложение З**

|  |
| --- |
| .586  .model flat, stdcall  includelib libucrt.lib  includelib kernel32.lib  includelib ..\Debug\FuncLib.lib  ExitProcess PROTO:DWORD  EXTRN random: PROC  EXTRN toPow: PROC  EXTRN writestr: PROC  EXTRN writenum: PROC  EXTRN writenumline: PROC  EXTRN writestrline: PROC  EXTRN system\_pause:PROC  .stack 4096  .const  ZEROMESSAGE BYTE 'Ошибка:деление на ноль',0  OVERFLOWMESSAGE BYTE 'Ошибка:переполнение типа',0  countsumcalculating\_sum BYTE 'calculating sum', 0  coutsomethcalculating\_combocalculating\_combo BYTE 'calculating combo', 0  main4 DWORD 4  main5 DWORD 5  main0xf DWORD 0  main10 DWORD 10  maincounting\_more\_then\_10 BYTE 'counting more then 10', 0  maincounting\_less\_then\_10 BYTE 'counting less then 10', 0  mainl BYTE 'l', 0  main0 DWORD 0  main3 DWORD 3  main2 DWORD 2  main1 DWORD 1  mainnew\_calculating BYTE 'new calculating', 0  main7 DWORD 7  main9 DWORD 9  main6 DWORD 6  .data  mainoperation1 DWORD ?  mainhexvalue1 DWORD ?  maincounter1 DWORD ?  mainmessage1 DWORD ?  .code  countsum PROC countsumn1 : DWORD , countsumm1 : DWORD  push OFFSET countsumcalculating\_sum  pop eax  push eax  call writestrline  push countsumn1  push countsumm1  pop eax  pop ebx  add eax,ebx  push eax  jc OVERFLOW  pop eax    ret  ZEROERROR:  push OFFSET ZEROMESSAGE  call writestrline  push -1  call ExitProcess  OVERFLOW:  push OFFSET OVERFLOWMESSAGE  call writestrline  push -1  call ExitProcess  countsum ENDP  coutsomethingelse PROC coutsomethnn1 : DWORD , coutsomethmm1 : DWORD  push OFFSET coutsomethcalculating\_combocalculating\_combo  pop eax  push eax  call writestrline  push coutsomethnn1  push coutsomethmm1  pop eax  pop ebx  add eax,ebx  push eax  jc OVERFLOW  push coutsomethnn1  push coutsomethmm1  call countsum  push eax  pop eax  pop ebx  add eax,ebx  push eax  jc OVERFLOW  pop eax    ret  ZEROERROR:  push OFFSET ZEROMESSAGE  call writestrline  push -1  call ExitProcess  OVERFLOW:  push OFFSET OVERFLOWMESSAGE  call writestrline  push -1  call ExitProcess  coutsomethingelse ENDP  main PROC  push main4  push main5  call coutsomethingelse  push eax  pop eax  push eax  pop mainoperation1  push mainoperation1  pop eax  push eax  call writenumline  push main0xf  pop eax  push eax  pop mainhexvalue1  push mainhexvalue1  pop eax  push eax  call writenumline  push mainoperation1  push main10  pop ebx  pop eax  cmp eax, ebx  jb SKIP19  push OFFSET maincounting\_more\_then\_10  pop eax  push eax  call writestrline  jae SKIPELSE23  SKIP19:  push OFFSET maincounting\_less\_then\_10  pop eax  push eax  call writestrline  push OFFSET mainl  pop eax  push eax  call writestrline  SKIPELSE23:  push main0  pop eax  push eax  pop maincounter1  push main5  call random  pop ecx  push eax  push main3  push main2  call toPow  pop ecx  push eax  pop ebx  pop eax  mul ebx  push eax  jc OVERFLOW  pop eax  push eax  pop mainoperation1  push mainoperation1  push main1  pop ebx  pop eax  sub eax,ebx  push eax  jc OVERFLOW  push main3  pop ebx  pop eax  test ebx, ebx  jz ZEROERROR  cdq  div ebx  push eax  jc OVERFLOW  pop eax  push eax  pop mainoperation1  push mainoperation1  pop eax  push eax  call writenumline  push OFFSET mainnew\_calculating  pop eax  push eax  pop mainmessage1  push mainoperation1  push main7  pop ebx  pop eax  test ebx, ebx  jz ZEROERROR  cdq  div ebx  push edx  jc OVERFLOW  pop eax  push eax  pop mainoperation1  push main10  push main10  pop ebx  pop eax  cmp eax, ebx  jne SKIP38  push main10  pop eax  push eax  call writenumline  SKIP38:  push main9  push main10  pop ebx  pop eax  cmp eax, ebx  je SKIP42  push main9  pop eax  push eax  call writenumline  SKIP42:  push mainoperation1  push main5  pop ebx  pop eax  cmp eax, ebx  jae SKIP46  push mainmessage1  pop eax  push eax  call writestrline  SKIP46:  push mainoperation1  push main6  pop ebx  pop eax  cmp eax, ebx  jbe SKIP50  push mainmessage1  pop eax  push eax  call writestrline  SKIP50:  push main0  pop eax  push eax  call system\_pause  call ExitProcess  ZEROERROR:  push OFFSET ZEROMESSAGE  call writestrline  push -1  call ExitProcess  OVERFLOW:  push OFFSET OVERFLOWMESSAGE  call writestrline  push -1  call ExitProcess  main ENDP  END main |

Листинг 9 – Генерация кода